Optimizing Message Sends in Object Oriented Languages through Type Invariant Region Analysis *

Mark Leair[1] and Santosh Pande[2]

[1] The Portland Group, Inc., 9150 SW Pioneer Ct. #H, Wilsonville, OR 97070 USA
mleair@pgroup.com
[2] University of Cincinnati Dept of ECECS, P.O. 210030 Cincinnati, OH 45221 USA
santosh@ececs.uc.edu

Abstract. Compilers for Object-Oriented Languages optimize dynamic message sends through run-time type testing when they cannot precisely infer the message receiver type. While run-time type testing optimizes dynamic messages, it still results in redundant type testing. We propose a flow-sensitive analysis which calculates regions in a program where an object's type is statically unknown but invariant. The type check is hoisted at the entry point of a region and its results are shared over all dynamic call-sites in the region. We implemented this optimization in the Vortex compiler and demonstrate its effectiveness on five large Cecil benchmarks. We observed an average speed-up of 9% and 18% speed-up in the best case.

1 Introduction and Related Work

Typical programs written in Object-Oriented Programming Languages (OOPLs) result in a large number of dynamic message sends due to lack of static type information. Dynamic messages sends can be optimized in OOPLs by using a compile-time interprocedural *type inference* [24] [22] [21] [23] [3] [12]. However, at best only 60% of the message receiver types can be precisely inferred[1].

When a type inferencer cannot statically bind a message send, the compiler inserts some sort of run-time type test. If the set of possible receiver types for a message send is small, the compiler will inline a series of type tests [2]. Otherwise, the compiler generates a dynamic run-time type test through a (polymorphic) inline cache [14] [19].

In all the approaches above, some sort of run-time type check is required when a dynamic message cannot be precisely inferred. However, neither of these

* This research is partially supported by NSF under grant # CCR 9696129

[1] DeFouw et al. [13] showed that only 40-60% of the dynamic call-sites in the Cecil [6] OOPL could be inferred. Agesen and Hölzle [2] conducted a similar study on the Self [25] OOPL and found that 30-40% of all dispatches remained in their tiny benchmarks, 50-60% in their small to medium benchmarks, and 25-60% in their large benchmarks.

techniques take into account type check redundancy when a type of a receiver object does not mutate (or change) within a large program region. The fact is, not all method calls are dynamically bound during run-time. Grove et al. [18] showed that 50% of the dynamic dispatches in their Cecil programs had a single receiver class. This would indicate that many dynamically typed objects, although statically unknown in their type binding, have a fixed type during a particular region in the program. In these cases, type tests are only needed at the first use of the object within the region. Run-time type tests or cache probes for subsequent uses of these objects are extraneous.

A partial redundancy elimination (PRE) of redundant type tests can be performed. One such optimization is *splitting* [10]. While splitting can eliminate redundant type tests, code between the first test and redundant test must be duplicated. If the benefiting code is far away, the increase in code size may outweigh the benefits of splitting. This would indicate the need for a new technique based on PRE that can be applied to large regions of code. Our framework, presented in the next section, is motivated by this fact.

2 Type Invariant Region Analysis

In order to eliminate unneeded type tests, one could perform a static analysis to determine the program region where the type (although statically unknown) is invariant once it is dynamically fixed. Methods dispatched on objects whose types are fixed in a particular program region could be directly called; saving on the overhead of type tests and/or dynamic dispatches. To maximize the call-sites that are covered by the type invariant region, one could take the transitive closure of the object's reaching definitions and use a flow-sensitive analysis for better precision. After calculating this information, we can perform one type test and share its results for all call-sites in the region[2]. The end result is fewer type checks and more direct calls in a given program. Because we generate more direct calls, the performance greatly improves. The following motivating example shows the benefit of our approach.

2.1 Motivating Example

Figure 1 shows part of a list library written in Cecil. The method *reverse* implements a recursive list reversal on a list with data polymophism[3] (the integer implementation is shown). Because the second parameter ($Alist$) has data polymophism, the compiler cannot statically bind the tail recursive call.

One observation that can be made from this example is that the types of the formal parameters are unknown at compile-time, but known at the first invocation of the method. In other words, they do not mutate across recursive invocations. Most OO languages with functional flavor (such as Cecil) support

[2] A more formal definition of region is given later.

[3] The *m_list* object can store a heterogeneous list of integers, floats, strings, etc.

this model since formal parameters are immutable [6]. This allows us to statically define a region on the *Alist* formal parameter as illustrated in figure 2. Next, we can fix the targets of each call-site in the region at the first recursive invocation of reverse. Because the type of Alist is immutable, we are guaranteed that the targets for the Alist region will not change across recursive invocations. This allows us to cache the method targets and save on the method lookup overhead across recursive invocations leading to better run-time performance. We also observe that in many typical applications, the parameter types do not always mutate across *multiple* invocations of the same method at *different* call-sites.

```
method reverse(front@:int, Alist@:m_list[int]) : m_list[int] {
     if(Alist.is_empty, { ^Alist; });
     let var Blist := reverse(first(Alist),rest(Alist));
     Blist.add_first(front);
     Blist;
}
```

Fig. 1. Motivating example Cecil code

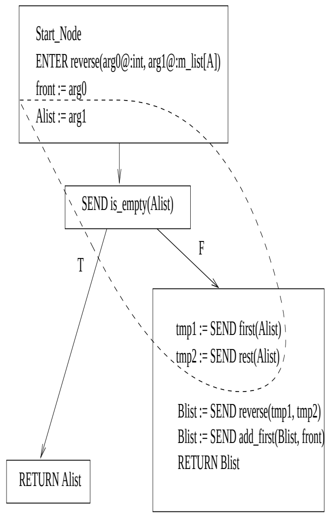

Fig. 2. Motivating Example Control Flow Graph with Corresponding Type Invariant Region for Parameter Alist.

2.2 Terms and Definitions

- A **control flow graph** (CFG) is a directed graph $G = (N, E, START, END)$, where N is the set of nodes, E is the set of flow edges, $START \in N$, and $END \in N$.
- A **definition** of an object instance x is a statement that assigns, or may assign, a value (as well as type) to x.
- When we make an assignment to an object instance x, the set of *definitions* that *may* determine the *type* and *value* of x are called the set of **reaching definitions** for x.
- The **receiver set** of a message send is the set of object definitions that determine the destination of the message. This set consists of 1-tuples for **singly-dispatched** message sends and n-tuples of object definitions for **multiple-dispatched** (or multi-method) [8] message sends.
- The program **call graph** is a CFG that represents the (inter-procedural) calling relationships between the program's procedures.
- The **type binding** (or simply type) of any object used in the receiver set for a singly- or multiple-dispatched message send is determined solely by a data-flow (e.g. an initialization, assignment, side-effect through a message send, etc.) [4].

2.3 Formal Framework

Our goal is to detect the portion of a method's CFG in which the type of a call-site's dispatching argument is invariant. This portion of the CFG is referred to as the *type invariant region*. After detecting the type invariant region of an object, run-time type tests can be hoisted to the top of the region and shared with all call-sites that are dispatched on the type of the object.

For purposes of efficient code hoisting, we require a flow-sensitive analysis. We are only interested in dynamic call-sites whose types *must* remain invariant based on the type of the formal parameter, not those that *may* remain invariant (computed by a flow-insensitive analysis).

Definition 1. *region(n) = {A Method call-site p such that the callee of p can be uniquely inferred knowing the type binding of formal parameter n}*

Consider the method call-site p(a, b, c) within an outer call q(n) in figure 3. Assume p(a,b,c) is dispatched on arguments a, b and c. Our goal is to determine if the call-site p belongs to the region(n). If the types of a, b and c can be uniquely inferred knowing the type binding of n, then we can deduce the callee of method p and avoid a dynamic dispatch. We now propose a property that must be satisfied in order to determine the region of a formal parameter.

Lemma 1. *Refer to figure 3. Let n^i denote a possible type binding of formal parameter n; a_j denote a reaching definition of a; b_k denote a reaching definition of b; and c_r denote a reaching definition of c to a call-site p(a,b,c).*

A call-site p belongs to the region(n) only if $\forall$ a_j, b_k, c_r, type(n^i) $\Rightarrow$ type(a_j) and type(n^i) $\Rightarrow$ type(b_k) and type(n^i) $\Rightarrow$ type(c_r) such that the inferred type bindings for a, b and c lead to a single callee of p.

```
method q(n) : void {
...
p(a, b, c);
...
}
```

Fig. 3. Formal Framework Motivating Example

Proof:

It is obvious that if we can infer the types of a, b, and c in p(a, b, c) by using a type binding of n that leads to a single callee for p, then we can uniquely bind the dispatch of p. In order for this to be true, all the reaching definitions of parameters a, b and c must be type deducible for a given type binding n_i of n and must lead to a single callee for p.

Definition 2. *typededucible(n^i) = {The set of all definitions d_i such that the type of d_i can be uniquely inferred using a given type binding n^i}*

Corollary 1. *p(a, b, c) is in the region of n if for a given type binding n^i of n, $\forall a_j$, b_k and $c_r \in$ typededucible(n^i) leading to a single callee for p where a_j, b_k and c_r represent reaching definitions of a,b and c respectively at call-site p.*

In its full generality, computing typededucible(n^i) and region(n^i) poses the problem of *flow-sensitive* type inferencing using reaching definitions. Several algorithms for inferring types and constructing call graphs for OOPLs have been proposed, however they are *flow-insensitive* approaches [4] [3]. *Flow-sensitive* analysis is needed to avoid a loss of precision in our framework. In order to build a flow-sensitive call graph, reaching definitions need to be calculated. However, solving such a problem in the interprocedural scope can have a considerable complexity as shown by Agesen [4]. Morever, there is no empirical data yet that demonstrates the tractability of flow-sensitive call graph construction using reaching definitions [1]. Thus, we limit our analysis to the intraprocedural scope. Also we propose a framework based on reaching definitions of values rather than types. This allows us to avoid type inferencing. Although this approach is certainly less precise, we contend that in the intraprocedural sense the loss of precision is quite tolerable due to the limited scopes and type locality of the values involved.

In our value based approach, we use the closure of reaching definitions of formal parameter n. We introduce a concept of reaching_defs_copy(n).

Definition 3. *reaching_defs_copy(n) is the set of all values that are copies of reaching definitions of n within the callers' scope of method q(n).*

[4] Reaching definitions and the call graph must be built simultaneously [1]. Unfortunately this adds a layer of complexity to the already complex iterative refinement that resolves the type and call graph mutually dependent problems [17] [16]. Agesen also warns that it easy to find code examples with inferior asymptotic precision [1].

Thus, $k \in reaching\_defs\_copy(n)$ if and only if either there exists an assignment k = n or k = r where, $r \in reaching\_defs\_copy(n)$.

Using these definitions, we can restate the region of n as follows:

Corollary 2. *p(a, b, c) $\in$ region(n) if $\forall a_j$, b_k and $c_r \in$ reaching_defs_copy(n) leading to a single callee for p where a_j, b_k and c_r represent reaching definitions of a,b and c respectively at call-site p.*

Proof:

It is obvious that the set reaching_defs_copy(n) $\subseteq \bigcup_i typededucible(n^i)$, thus satisfies corollary 1.

In order to effectively use corollary 2, we must be able to include all call-sites that could be inferred using the transitive closure of the set $reaching\_defs\_copy(n)$. Finally, if the outer call-site q(n) had multiple dispatching arguments (such as in q(n,m)) corollary 2 still holds, except that we now have a definition of the region in terms of a cartesian product of type bindings n and m. The above definitions could be trivially extended to these cases.

Lemma 2. *The type for an object definition x is said to be invariant from a program point y to a program point z if and only if there does not exist a path from y to z that kills definition x.*

Proof:

When considering objects that are used in singly- or multiple-dispatched message sends, the type for an object is determined solely on a data-flow (e.g., an initialization, assignment, side-effect through a message send, etc.) [4]. In order for x to mutate, there must exist a path from y to z, such that definition x gets killed. Conversely, if there does not exist a redefinition of x along any path between y and z, then the type for x must remain invariant between y and z if and only if type is determined solely by a data-flow.

Definition 4. *Using the definitions of x, y, and z from Lemma 2, we describe the type invariant region for x as all the paths from y to z if and only if lemma 2 holds.*

The above lemma and definition for *type invariant region* is applicable for singly- or multiple-dispatched message sends. *Predicate dispatching* [7] [15], another (although less common) dispatching technique, is not considered here [5].

2.4 Algorithms for Type Invariant Region Detection

The implementation algorithms are presented in figures 4 and 5. The first algorithm takes as input $DefUse(X)$ which is calculated using a standard du chain algorithm [5]. Next, we calculate region_reaching_defs_copy($region\_id$) in figure 4. In the first loop we are seeding region_reaching_defs_copy(n) with all

[5] Determining the type-invariant region for an object used in predicate dispatching is undecidable without loss of generality. A proof was omitted due to space limitations. The interested reader may request the proof from the authors.

immutable objects[6]. In the second loop we calculate their transitive closure. After region_reaching_defs_copy(region_id) is calculated, we can calculate region(n) (definition 1) in figure 5. We use a table called *region_table* that maps a receiver set to a set of message sends. We initially seed region_table with the range of region_reaching_defs_copy. Next, we traverse a CFG and examine each dynamic message send. Whenever we encounter a dynamic message send in which its receiver set is in the domain of region_table, we add it to the appropriate message set. After traversing the CFG, region(n) is defined.

Algorithm 1: Compute_Region_Reaching_defs_copy(region_cnt)

Input: $DefUse(X)$,
$DefUse(X)$ is a table that maps the object definition X to a set of reaching object definitions used by X
Output: region_reaching_defs_copy($region\_cnt$),
table $region\_reaching\_defs\_copy(region\_id)$ maps $region\_id \in integer \rightarrow reaching\_defs\_copy(n)$,
region_cnt = the number of regions detected in a CFG and
table $region\_reaching\_defs\_copy$ maps $(1 \leq region\_id \leq region\_cnt) \rightarrow reaching\_defs\_copy(n)$.

```
change := true;
region_cnt := 1;

for each Z ∈ Domain(DefUse(X)) do
        if( Z is an immutable object) then
            region_reaching_defs_copy(region_cnt) := region_reaching_defs_copy(region_cnt) ∪Z;
            increment region_cnt;
        end if
end do

while( change = true ) do
        change := false;
        for each Z ∈ Domain(DefUse(X)) do
            let Y := DefUse(Z);
            if( ¬(Z ∈ Y) ) then
              for each i ∈ region_reaching_defs_copy(n) do
                if( Y ⊆ region_reaching_defs_copy(i) ) then
                  region_reaching_defs_copy(i) := region_reaching_defs_copy(i) ∪Z;
                  change := true;
                end if
              end do
            end if
        end do
end do
```

Fig. 4. Algorithm 1: Compute region_reaching_defs_copy(region_cnt)

[6] In Cecil these include all formal parameters, user-specified immutables, and compiler-generated temporaries. All other objects (e.g., variables) in Cecil are mutable [6].

Algorithm 2: ComputeRegions

Input: The control flow graph (CFG), region_reaching_defs_copy($region\_cnt$)
(computed in Algorithm 1).
Output: region(n) (definition 1).

let $message\_set$ be a set of nodes $N \in CFG$ such that N is a dynamic message send;
let $region\_table$ be a table that maps $receiver\_set \rightarrow message\_set$;

for each i $\in$ region_reaching_defs_copy(n) do
    Domain(region_table) := Domain(region_table) $\cup$ reaching_defs_copy(i)

for each node $N \in message\_set$ do begin
    let $R := receiver\_set(N)$;
    if($R \in Domain(region\_table)$) then
        $region\_table(R) := region\_table(R) \cup N$;
    else
        if($\forall S \in R \exists i \in Range(region\_reaching\_defs\_copy(region\_cnt))$ such that $S \subseteq i$) then
            $region\_table(R) := N$;
end;
let $region(n) := region\_table$;

Fig. 5. Algorithm 2: Compute region(n)

3 Implementation

We implemented our Type Invariance Region Analysis (TIRA) framework in Vortex [11], an optimizing OOPL back-end with front-ends for Cecil, Modula-3, C++, and Java. We implemented our framework as the last optimizing pass. The optimizations that precede our analysis are: Static Class Analysis [9], Iterative Class Analysis [10] [20], Class Hierarchy Analysis [12], Inlining [9], Splitting [10], and Common Subexpression Elimination [5]. The rationale for placing our analysis last is that we will optimize only those message sends that the earlier optimizations could not remove. Moreover, placing it after inlining gives our framework potentially larger regions to work with. Because our optimization uses reaching definitions of values rather than types, we do not need an interprocedural class analysis. We also focus on optimizing regions with more than one call-site because a region with one call-site is equivalent to an inline cache or run-time type test[7].

In figure 2 $region(Alist)$ is defined after the statement $Alist := arg1$. We call this statement the *region entry point.* During code generation, we generate

[7] A region with one call-site inside a loop may benefit, but most loops in our benchmarks were not implemented inline. Instead, they were implemented with a closure passed into a loop method.

run-time code to perform the *region type test* at the region entry point. We also generate static message sends for each region call-site.

Each region has a *region cache* variable that stores the most recent type binding for n in $region(n)$. Each region call-site has a cache that stores the most recent message target. At the region entry point, we generate a comparison between the type of n and the type stored in the region cache. Whenever the type binding for n differs from the region cache, we perform a dynamic dispatch for each region call-site. When n does not differ, it is safe to reuse the cached message targets for each region call-site. Figure 6 illustrates the run-time code for singly-dispatched regions. If the region consists of multiple-dispatched methods, a region cache for each dispatching argument is required.

```
OOP X;                                        /*assume some object X*/
static OOP_MAP * REGION_CACHE_TYPE = NULL;    /*region cache*/
static void (*CALL_SITE_CACHE_1) (ARG_LIST); /*call-site cache 1*/
...
static void (*CALL_SITE_CACHE_N) (ARG_LIST); /*call-site cache N*/
/**** The code below is generated at the region entry point ****/
OOP_Map * Xtype = X->type();
if ( Xtype != REGION_CACHE_TYPE ){
      CALL_SITE_CACHE_1 = methodLookup(selector_1); /*lookup method 1*/
      ...
      CALL_SITE_CACHE_N = methodLookup(selector_N); /*lookup method N*/
      REGION_CACHE_TYPE = Xtype;       /*set region cache to new type*/
}
...
/***** Region call-sites below *****/
STATIC_SEND((*CALL_SITE_CACHE_1)(args)); /*directly call cached method 1*/
...
STATIC_SEND((*CALL_SITE_CACHE_N)(args)); /*directly call cached method N*/
```

Fig. 6. TIRA Run-Time Pseudo-code

3.1 Performance Evaluation

Table 1 summarizes our results [8]. The *Regions Detected* column breaks down the number of regions and sizes (e.g., number of call-sites) found by our framework. The *Regions Executed* column lists the number of regions and their sizes that were used at run-time. The *Hit Rate* column measures region cache performance. When the region cache misses, our run-time must perform method lookups for each region call-site. On a cache hit, each call-site directly invokes the method

[8] Benchmarking was performed on an unloaded 333 MHz UltraSPARC 5/10 workstation with 256MB of RAM.

stored in its call-site cache. The *Type Tests Avoided* column lists the number of receiver type comparisons avoided based on the region size. *Time Base* is the execution time for the benchmark compiled with the default full optimization level in Vortex. *Time TIRA* is the execution time for the benchmark compiled with full optimization and our TIRA framework.

The majority of the regions detected were found in Cecil's standard library[9]. This probably explains the high cache hit rates as library methods are generally invoked at the end of a polymorphic call chain. This reveals a nice application for TIRA. TIRA can be used to optimize separately compiled libraries when no other type information is available.

4 Conclusion

In this work we developed a framework that augments existing intra and inter-procedural type analyses for OOPLs. Because our analysis works off of reaching definition values rather than types, our analysis can optimize message sends in which no available type information is present. This is especially useful for the separate compilation of library routines where the types of the incoming formal parameters are statically unknown but fixed at run-time. Next, we demonstrated in our benchmarks that most of the messages that we optimized had a single receiver type. This is evident with the high region cache hit rates and large regions found. Finally, we used our type invariant region analysis to remove redundant receiver type tests and improve performance. Average performance improvement was 9% with 18% in the best case.

References

1. Ole Agesen, *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD Dissertation. Stanford University, December 1995.
2. Ole Agesen and Urs Hölzle, *Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object Oriented-Languages.* In OOPSLA '95, Tenth Conference on Object-Oriented Programming, Systems, Languages, and Applications, Austin, TX, October 1995.
3. Ole Agesen, *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism.* In ECOOP '95, Ninth European Conference on Object-Oriented Programming, Arhus, Denmark, August 1995.
4. Ole Agesen, *Constraint-Based Type Inference and Parametric Polymorphism.* IN SAS '94, First International Static Analysis Symposium, Springer-Verlog LNCS 864, 1994.
5. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.
6. Craig Chambers, *The Cecil Language: Specification and Rationale version 2.1.* Technical Report TR-97-03-31, Department of Computer Science and Engineering. University of Washington, March 1997.

[9] All 52 of deltablue's regions were in the library and 275 were in the library for the other four benchmarks.

Benchmark Performance									
Name	**Description**	**Lines**[a]	**Regions Detected** (size:qty)	**Regions Executed** (size:qty)	**Hit Rate** (%)	**Type Tests Avoided** (size:qty)	**Time Base**	**Time TIRA**	**Speed up** (%)
delta-blue	Constraint solver		2:36 3:12 4:3 5:1	2:2 3:1 4:0 5:1		2:14,624 3:264 5:5,764			
total		**650**	**52**	**4**	**99**	**20,652**	**147ms**	**140ms**	**5.00**
instr-sched	Global MIPS instruction scheduler		2:197 3:42 4:24 5:6 6:4 7:2 8:1	2:5 3:2 4:1 5:1 6:0 7:1 8:0		2:17,465 3:10,578 4:510 5:56,532 7:86,616			
total		**2.4K**	**276**	**10**	**98**	**171,701**	**2.24s**	**2.11s**	**6.16**
type check	Typechercker for old Cecil type system		2:208 3:46 4:23 5:6 6:4 7:2 8:1	2:40 3:14 4:3 5:3 6:1 7:2 8:0		2:296,588 3:42,822 4:5,304 5:228,996 6:10 7:99,096			
total		**20K**	**290**	**63**	**97**	**672,649**	**16s**	**14s**	**14.29**
new-tc	Typechecker for new Cecil type system		2:216 3:48 4:25 5:6 6:5 7:2 8:1	2:43 3:14 4:6 5:3 6:2 7:2 8:0		2:342,489 3:79,352 4:43,551 5:72,904 6:9,965 7:190,434			
total		**23.5K**	**303**	**70**	**97**	**738,695**	**13s**	**11s**	**18.18**
comp	Old version of the Vortex Compiler		2:291 3:63 4:29 5:6 6:5 7:3 8:1 10:1	2:52 3:16 4:6 5:3 6:1 7:2 8:0 10:0		2:6,043,952 3:323,680 4:85,089 5:7,758,632 6:39,415 7:3,210,312			
total		**50K**	**399**	**80**	**90**	**17,461,080**	**566s**	**552s**	**2.57**

Table 1. Performance Evaluation

[a]Does not include an 11,000 line standard library.

7. Craig Chambers, *Predicate Classes.* In ECOOP '93 Conference Proceedings. Kaiserslautern, Germany. July, 1993.
8. Craig Chambers, *Object-Oriented Multi-Methods in Cecil.* In ECOOP '92 Conference Proceedings, Utrecht, the Netherlands, June/July 1992.
9. Craig Chambers, *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992.
10. Craig Chambers and David Ungar. *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs.* PLDI '91, June 1990.
11. Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers, *Vortex: An Optimizing Compiler for Object-Oriented Languages.* In OOPSLA '96, 11th Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Jose, CA, October 1996.
12. Jeffrey Dean, David Grove, and Craig Chambers, *Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis.* In *Proceedings* ECOOP '95, August 1995.
13. Greg DeFouw, David Grove, and Craig Chambers, *Fast Interprocedural Class Analysis.* In *Proceedings* POPL '98, San Diego. CA, January 1998.
14. L. Peter Deutsch and Alan Schiffman, *Efficient Implementation of the Smalltalk-80 System.* In Proceedings of the 11th. Symposium on the Principles of Programming Languages, Salt Lake City, UT, 1984.

15. Michael Ernst, Craig Kaplan, and Craig Chambers, *Predicate Dispatching: A Unified Theory of Dispatch.* In ECOOP '98 Proceedings. Brussels, Belgium, July 1998.
16. David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers, *Call Graph Construction in Object-Oriented Languages.* In OOPSLA '97, 12th Conference on Object-Oriented Programming, Systems, Languages, and Applications, Atlanta, GA, October 1997.
17. David Grove. *The Impact of Interprocedural Class Analysis on Optimization.* In CASCON '95 Conference Proceedings. Toronto, Canada, November, 1995.
18. David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers, *Profile-Guided Receiver Class Prediction.* OOPSLA '95, Austin, TX, October 1995.
19. Urs Hölzle, Craig Chambers, and David Ungar, *Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches.* In ECOOP '91 Conference Proceedings, Geneva, 1991. Published as Springer Verlag Lecture Notes in Computer Science 512, Springer Verlag, Berlin, 1991.
20. Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. *TS: An Optimizing Compiler for Smalltalk.* In OOPSLA '88 Conference Proceedings, October 1988.
21. Nicholas Oxhϕj, Jens Palsberg, and Michael I. Schwartzbach. *Making Type Inference Practical.* In ECOOP '92 Conference Proceedings, June/July 1992.
22. Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Inference.* In OOPSLA '91 Conference Proceedings, October 1991.
23. John B. Plevyak and Andrew A. Chien, *Precise Concrete Type Inference for Object-Oriented Programming Languages.* In OOPSLA '94, Ninth Conference on Object-Oriented Programming, Systems,Languages, and Applications, Portland, OR, October 1994.
24. Olin Shivers. *Control Flow Analysis in Scheme.* In PLDI '88 Conference Proceedings, June 1988.
25. David Ungar and Randall B. Smith. *Self: The Power of Simplicity.* In OOPSLA '87 Conference Proceedings. 1987.